Supermon User's Guide

The Supermon Developers

September 18, 2002

Chapter 1

Introduction

Supermon is a set of programs and APIs for gathering and managing monitoring data from large-scale clusters at high sampling rates. The basic supermon installation provides most system metrics of interest to cluster users and administrators. This includes CPU load, network traffic, disk activity, and memory information. Systems with supported sensor chipsets can also provide data such as core temperature and fan speed directly from hardware for physical status monitoring. In addition, user-level programs can use the *monhole* to take advantage of supermon's efficient data transport scheme for their own data. Examples of monhole usage include custom system monitors (for unusual operating systems or custom hardware sensors) and application level events.

Figure 1 illustrates an example supermon system architecture for a cluster. The data servers, `mon`, read data from `/proc/sys/supermon`; there is a single mon process running on each node (not processor) of the cluster. The data concentrator, `supermon`, sends requests and composes data from multiple sources; the sources may be mon programs or other data concentrators. Finally, user level programs may add data to the data stream by connecting to the monhole.

Both `mon` and `supermon` use a hierarchical data protocol based on LISP-style symbolic expressions (s-expressions). S-expressions are *structurally* self-describing, that is, no meta-data is needed in regards to the semantics or typing of the contents of the data stream. S-expressions are also naturally composable, thus greatly simplifying the job of `supermon`, the data concentrator. The format of this protocol is described in Chapter 3.

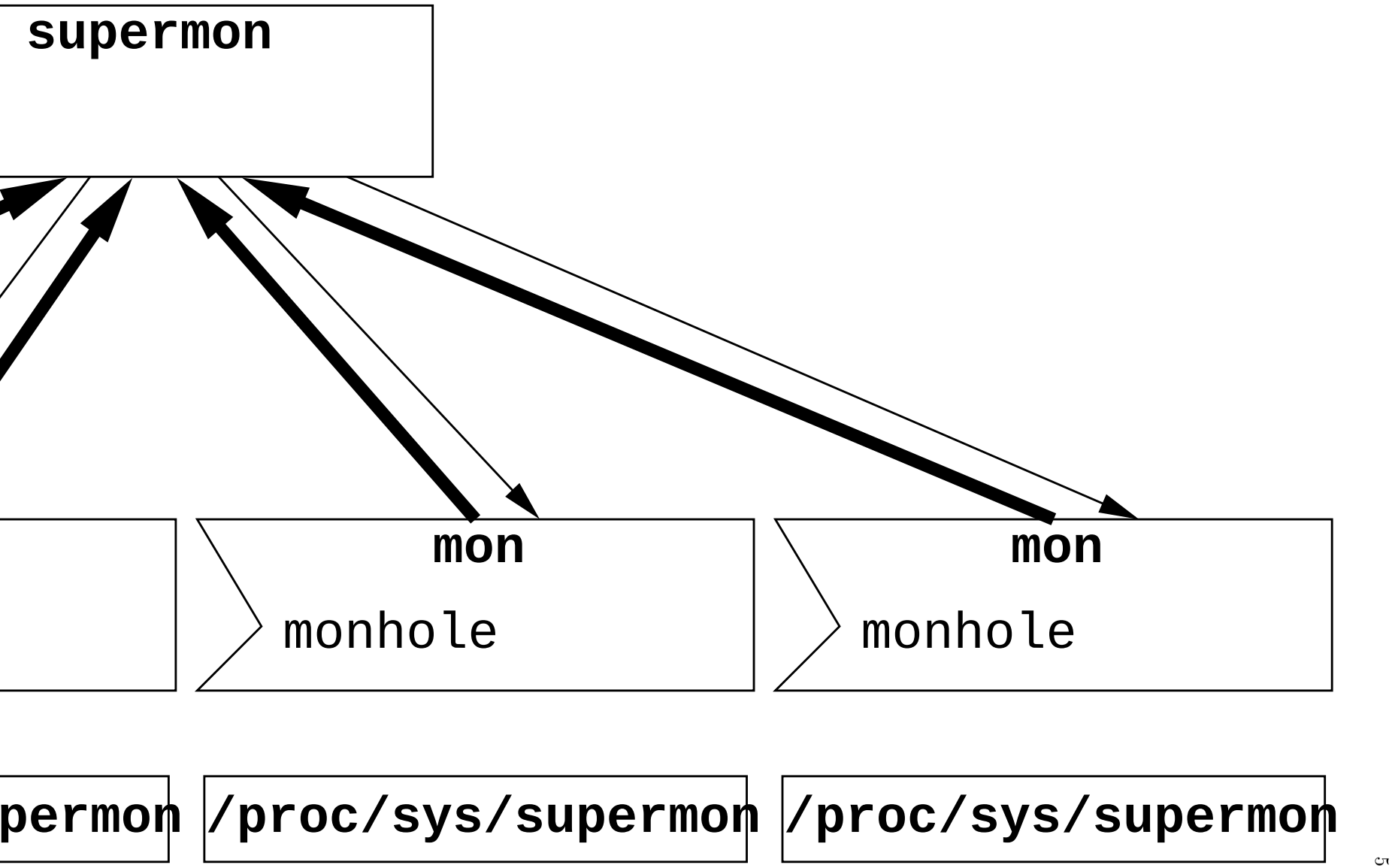
Rather than the data server controlling sampling rates, data sampling is dictated by the clients themselves. This allows supermon users to have some control over perturbation of cluster performance due to monitoring. When no users or programs are gathering monitoring data, the supermon system quietly moves into the background allowing application tasks to have full priority on the cluster. Chapter 5 describes ways to tune supermon with a reasonable sampling rate for a particular cluster.

Mon, supermon, monhole, and the s-expression library require basic UNIX features such as TCP sockets, Unix domain sockets (for monhole), and basic signal and sleeping features. The only system-specific portion of code in Supermon is the Linux kernel module used to provide the data index and sampling entries in `/proc/sys/supermon`.

Supermon is part of a larger suite of cluster management and application software called *Clustermatic*. A core technology in Clustermatic is *BProc*, Linux kernel modifications that result in greatly improved cluster process management. Though supermon can be used on any type of cluster, this manual focuses on operation in the BProc Linux environment. From the standpoint of supermon, the only difference is process start up and shut down which easily translates

into more traditional cluster environments. For more information on BProc and Clustermatic, see `http://www.clustermatic.org`.

Currently supermon has been extensively tested under Linux. Additional work has been done on MacOSX and Irix, so porting issues with these platforms should be minimal (with the exception of the kernel module). Porting efforts will most likely revolve around creating a data source like the Linux kernel module for the specific operating system being targetted.


supermon
mon
monhole
mon
monhole
permon
/proc/sys/supermon
/proc/sys/supermon

Chapter 2

Getting Started

Building and running supermon is fairly straightforward. If your system does not already have the supermon kernel module installed, you (or your system administrator) will need to install it. Both mon and supermon can be run by unpriveliged users. Supermon is installed by default for a *Clustermatic* software installation and other clustering packages that are based on BProc and Supermon.

2.1 Building supermon

First, run the provided `configure` script to generate the makefiles and perform any system specific configuration that is required.

```
% ./configure
```

Currently, configure is only used to detect the system type and find the C and C++ compilers. After successfully finishing, run `make` to build the supermon package.

```
% make
```

This command builds the `mon` and `supermon` servers, the s-expression library used by the servers and clients, and a set of test programs in the `bench` directory for benchmarking. By default, mon expects data to be provided by two entries in `/proc/sys/supermon`. We provide a Linux kernel module for this purpose. To build the kernel module, first figure out what kernel version you are running on the nodes to be monitored. If the cluster is a BProc-based cluster, the following command issued from the front end will return that information.

```
% bpsh nodenum uname -a
```

Assuming that your source tree for that kernel version is located in the `linux-`*`kernel_version`* subdirectory of `/usr/src`, then the following command can be executed from within the `kernel` subdirectory to build the kernel module. For example, we keep kernel version 2.4.18 for our cluster nodes in the directory */usr/src/linux-2.4.18-bproc/*. In cases where the nodes are heterogeneous,

kernel modules must be built against each version of the kernel found on the nodes. After each build, the `supermon_proc.o` binary should be moved to the nodes or stored in a safe place since it will get overwritten when running make for subsequent kernel versions.

```
% make LINUX=/usr/src/linux-kernel_version
```

2.2 Running supermon

The basic supermon system requires three steps to start once it has been built. First, the nodes that are to be monitored must have the supermon kernel module installed to provide the necessary entries in `/proc` for gathering the raw data. Second, the nodes must also have a single instance of the `mon` server running to provide the socket for clients to connect to to sample data. Finally, a `supermon` server must be started on either a node, front-end, or some other system to gather data from all nodes running `mon` and return it as a single cluster image.

Inserting the kernel module must be performed as `root`. Make sure that the `.o` file built in the `kernel` subdirectory has been copied out to all of the nodes. For most purposes, copying it to `/tmp` on each node is sufficient. Once it is out on the nodes, simply run `/sbin/insmod` as root to insert the kernel module. On a BProc-based cluster, the following command will do the job.

```
# bpsh -a /sbin/insmod /tmp/supermon_proc.o
```

Note that this is the only part of the process that must be run as root. If supermon is going to be used frequently, we recommend adding the module to the node startup process so that it is loaded at boot time.

Starting the mon processes is very simple, as it requires only running the `mon` server in the background on the nodes to monitor. Note that on BProc-based systems, the `-N` option must be included on the `bpsh` command to turn off I/O forwarding.

```
% bpsh -aN mon &
```

At this point, we simply need to start a `supermon` server to connect to each node on port 2709 (registered in `/etc/services` for supermon) and gather data into a single cluster sample image. The command line format is the port that supermon should open up to clients and the list of nodes to monitor and the corresponding ports to listen on (if different than the default, 2709). Assuming that mon has been run on its default port, we can start supermon on port 2710 on the current machine and monitor five nodes with the following command.

```
% supermon -p 2710 n0 n1 n2 n3 n4
```

For starting supermon on BProc clusters, a script called `runsupermon.pl` is provided. This script uses `bpstat` to determine which nodes are alive to build the proper supermon command line. Note that the nodes in the list are represented by their hostname or IP address, and may have different names than listed above.

```
((0x1 cpuinfo (nr (1 0xe04000a 0xf04000a 0x1004000a 0x1204000a))
              (user nice system))
 (0x2 avenrun (nr (1 0xe04000a 0xf04000a 0x1004000a 0x1204000a))
              (avenrun0 avenrun1 avenrun2))
 (0x4 paging (nr (1 0xe04000a 0xf04000a 0x1004000a 0x1204000a))
              (pgpgin pgpgout pswpin pswpout))
 (0x8 switch (nr (1 0xe04000a 0xf04000a 0x1004000a 0x1204000a))
              (switch))
 (0x10 time (nr (1 0xe04000a 0xf04000a 0x1004000a 0x1204000a))
              (timestamp jiffies))
 (0x20 netinfo (nr (2 0xe04000a 0xf04000a 0x1004000a 0x1204000a))
              (name rxbytes rxpackets rxerrs rxdrop rxfifo rxframe
               rxcompressed rxmulticast txbytes txpackets txerrs
               txdrop txfifo txcolls txcarrier txcompressed))
)
```

Figure 2.1: Sample `#` command output from a 4 node cluster.

2.3 Basic monitoring

The simplest way to examine monitoring data from either a `mon` or `supermon` server is to telnet to port 2709 (or whatever port is being used) on the host machine.

```
% telnet my_host 2709
```

At this point, one can manually issue commands and read the responses as plaintext s-expressions. The protocol for communicating with supermon and mon is described in detail in Chapter 3. Here we highlight examples of the `#` and `S` commands.

2.3.1 The # command

In Figure 2.1, the response to a single `#` command is shown. It is in the format of a single s-expression containing expressions for each category of data available. An expression describing a category shows the bitmask for that category, the name, a list of node IDs with the corresponding arity, and the list of variables in the category. We can see that this cluster has four nodes (by counting the IDs), each containing a single CPU (arity of 1 on the category *cpuinfo*), and two network interfaces (arity of 2 on the category *netinfo*). This is sufficient data for a client to traverse and interpret the output of the `S` command.

2.3.2 The S command

As illustrated in Figure 2.2, the structure of the `S` command result matches what was described in the `#` command output. Each expression starts with the appropriate bitmask and category name, followed by the node ID that provided the data for the given expression. This is then followed by a single expression containing a sequence of expressions representing each variable in the category. The variable expressions begin with the name of the variable followed by a sequence of atoms representing the data itself. The length of this sequence matches the arity of the category as indicated by the `#` command. For example, we see that for the `netinfo` category, where the arity was two in Figure 2.1, each variable has two values. We can see that one of them has not been used at all, and has zeros

for each of its values (in this case, this is the loopback device `lo`), while the other device has handled all of the network traffic.

It should be clear that for even a tiny cluster such as this, the amount of data that is returned in a single sample can be rather large. To minimize the amount of wasteful data included in a sample, clients can send together bitmasks indicating exactly which categories they want. For example, if we only cared about the `cpuinfo` and `switch` categories, we could send a mask of `0x9`. The result of this is shown in Figure 2.3.

In practice, manually issuing commands and interpreting s-expressions is tedious and impractical. A client program can be written in C, LISP, Perl, or any other language capable of connecting to the server over a socket and interpreting the resulting s-expressions. Supermon currently includes a very basic GTK+ based GUI client that will show a plot of data as it is being sampled. If users have more sophisticated needs, supermon includes a library (used by supermon itself) for parsing, traversing, and managing s-expressions from languages that do not support them already.

```
((0x1 cpuinfo 0xe04000a ((user 7) (nice 0) (system 299)))
 (0x2 avenrun 0xe04000a ((avenrun0 0) (avenrun1 0) (avenrun2 0)))
 (0x4 paging 0xe04000a ((pgpgin 0) (pgpgout 0) (pswpin 0) (pswpout 0)))
 (0x8 switch 0xe04000a ((switch 152107)))
 (0x10 time 0xe04000a ((timestamp 0x3d0fb8e7) (jiffies 16265062)))
 (0x20 netinfo 0xe04000a ((name lo eth0) (rxbytes 0 13209479) (rxpackets 0 21835)
                          (rxerrs 0 0) (rxdrop 0 0) (rxfifo 0 0) (rxframe 0 0)
                          (rxcompressed 0 0) (rxmulticast 0 0) (txbytes 0 3089066)
                          (txpackets 0 22662) (txerrs 0 0) (txdrop 0 0) (txfifo 0 0)
                          (txcolls 0 0) (txcarrier 0 0) (txcompressed 0 0)))
 (0x1 cpuinfo 0xf04000a ((user 6) (nice 0) (system 316)))
 (0x2 avenrun 0xf04000a ((avenrun0 0) (avenrun1 0) (avenrun2 0)))
 (0x4 paging 0xf04000a ((pgpgin 0) (pgpgout 0) (pswpin 0) (pswpout 0)))
 (0x8 switch 0xf04000a ((switch 152076)))
 (0x10 time 0xf04000a ((timestamp 0x3d0fc545) (jiffies 16265399)))
 (0x20 netinfo 0xf04000a ((name lo eth0) (rxbytes 0 13208393) (rxpackets 0 21824)
                          (rxerrs 0 0) (rxdrop 0 0) (rxfifo 0 0) (rxframe 0 0)
                          (rxcompressed 0 0) (rxmulticast 0 0) (txbytes 0 3088184)
                          (txpackets 0 22655) (txerrs 0 0) (txdrop 0 0) (txfifo 0 0)
                          (txcolls 0 0) (txcarrier 0 0) (txcompressed 0 0)))
 (0x1 cpuinfo 0x1004000a ((user 8) (nice 0) (system 327)))
 (0x2 avenrun 0x1004000a ((avenrun0 0) (avenrun1 0) (avenrun2 0)))
 (0x4 paging 0x1004000a ((pgpgin 0) (pgpgout 0) (pswpin 0) (pswpout 0)))
 (0x8 switch 0x1004000a ((switch 152594)))
 (0x10 time 0x1004000a ((timestamp 0x3d0fbba4) (jiffies 16265196)))
 (0x20 netinfo 0x1004000a ((name lo eth0) (rxbytes 0 13209533) (rxpackets 0 21832)
                          (rxerrs 0 0) (rxdrop 0 0) (rxfifo 0 0) (rxframe 0 0)
                          (rxcompressed 0 0) (rxmulticast 0 0) (txbytes 0 3089058)
                          (txpackets 0 22667) (txerrs 0 0) (txdrop 0 0) (txfifo 0 0)
                          (txcolls 0 0) (txcarrier 0 0) (txcompressed 0 0)))
 (0x1 cpuinfo 0x1204000a ((user 2) (nice 0) (system 305)))
 (0x2 avenrun 0x1204000a ((avenrun0 0) (avenrun1 0) (avenrun2 0)))
 (0x4 paging 0x1204000a ((pgpgin 0) (pgpgout 0) (pswpin 0) (pswpout 0)))
 (0x8 switch 0x1204000a ((switch 152065)))
 (0x10 time 0x1204000a ((timestamp 0x3d0fc16a) (jiffies 16265304)))
 (0x20 netinfo 0x1204000a ((name lo eth0) (rxbytes 0 13207955) (rxpackets 0 21817)
                          (rxerrs 0 0) (rxdrop 0 0) (rxfifo 0 0) (rxframe 0 0)
                          (rxcompressed 0 0) (rxmulticast 0 0) (txbytes 0 3088756)
                          (txpackets 0 22661) (txerrs 0 0) (txdrop 0 0) (txfifo 0 0)
                          (txcolls 0 0) (txcarrier 0 0) (txcompressed 0 0)))
)
```

Figure 2.2: Sample `S` command output from a 4 node cluster.

```
((0x1 cpuinfo 0xe04000a ((user 7) (nice 0) (system 299)))
 (0x8 switch 0xe04000a ((switch 152137)))
 (0x1 cpuinfo 0xf04000a ((user 6) (nice 0) (system 317)))
 (0x8 switch 0xf04000a ((switch 152108)))
 (0x1 cpuinfo 0x1004000a ((user 8) (nice 0) (system 327)))
 (0x8 switch 0x1004000a ((switch 152624)))
 (0x1 cpuinfo 0x1204000a ((user 2) (nice 0) (system 305)))
 (0x8 switch 0x1204000a ((switch 152097))) )
```

Figure 2.3: Sample `S` command after sending a bitmask of `0x9` for filtering.

Chapter 3

The supermon protocol

In this chapter we describe the protocol for communicating with supermon and interpreting supermon output.

3.1 Symbolic Expressions

A symbolic expression, or s-expression, is essentially a LISP-like expression such as `(a (b c))`. S-expressions are able to represent complex, structured data without requiring additional meta-data describing the structure. They are recursively defined: an s-expression is a list of either atoms or s-expressions. In the example above, the expression contains an atom "a" and an s-expression, which in turn contains two atoms, "b" and "c". They are simple, useful, and well understood.

Authors of clients will be able to either use standard LISP interpreters and compilers to deal with s-expressions, or can use a library provided with Supermon. This library provides interfaces in C, C++, and Python for parsing and traversing s-expressions from languages with no intrinsic support for them.

Future plans for 'filtermons', or supermon servers that interpret and filter data as it is being passed up to clients, involve embedding LISP interpreters into existing supermons to execute basic LISP operations while sampling occurs.

3.2 The # command

The `#` command is used to query a supermon or mon server in order to find out what data it provides and how it is structured. The set of data categories is provided, with their associated bitmask. For each category, the set of variables is included. Finally, the 'arity' of each node is given for each category. For example, if a cluster contained two types of nodes, one with two processors and the other with four, the `cpuinfo` category would list the node IDs with arity 2 (the dual processor machines), and the node IDs with arity 4. This allows the client to understand the structure of the expressions returned by the `S` command. The exact structure of the `#` command is shown in Figure 3.1.

The nodeid is a unique identifier for a single node in the cluster. Currently this ID corresponds to the IP address that supermon sees the node mon process listening on. The bitmask is a 32-bit non-zero hexadecimal number. The list of variable names has a slightly different meaning depending on its context. From a mon process, this list represents all data provided by that node in the

```
(mask category (n (n_1 nodeid ··· nodeid)
                  (n_2 nodeid ··· nodeid)
                  ···)
               (variable variable ··· variable))
```

Figure 3.1: The `#` command.

given category. Supermon on the other hand must determine the set of variables in a given category provided by all of the mon processes that it is connected to. So supermon will return the intersection of the variable sets from each mon process in each category. This guarantees that a sample will contain no missing data for nodes that potentially do not provide a particular metric.

3.3 The `S` command

Once a client has determined the data that is provided by a supermon or mon server, it uses the `S` command to retrieve samples of data. A single `S` command will yield a single sample of data. Like the result of the `#` command, a single sample is encapsulated in a single s-expression. This expression is a sequence of s-expressions containing a single category sample from a single node. The first three atoms of each category expression are the bitmask of the category, its name, and the node ID that provided the data. The final element of the expression is an expression containing the data itself. It is a sequence of expressions, each containing a single atom providing the name of a variable followed by a sequence of atoms containing the data itself. As explained earlier in Chapter 2, the number of elements following the variable name corresponds to the arity of the category as given in the `#` command.

3.3.1 Filtering

Filtering using bitmasks can reduce the size of a sample returned by the `S` command. The masks of each category to be sampled can be combined using a basic logical 'and' operator. To set a mask for filtering, a client simply must send the hexadecimal mask in plaintext to `supermon`or `mon`. The mask can contain, but is not required to have, a prefix of `0x`. Once a mask is set for a client, `supermon`and `mon`will maintain that mask for the duration of the connection to the client. A separate mask is maintained for each client. Although there is no command to reset the mask, simply sending a 32-bit number with all bits set (`0xffffffff`) will cause this to occur.

3.4 Changes in the system

In large clusters, individual nodes are frequently rebooted, crash, or have other issues that may cause a mon server to disappear temporarily. In addition, user-applications connect and disconnect to the monhole. Supermon is able to notice these changes, but it is up to the client program to act upon this change.

In either of these situations, it is possible that the `S` command output has change. These changes are flagged to client applications by sending a special s-expression with mask `0x0` indicating that either a client has gone away (`(0x0 died (<IP address>) (hostname <name>))`) or something has changed (`(0x0 changed (<IP address>))`). When the client receives notification of such changes, the data returned from `S` is not guaranteed to be valid until another `#` command is issued.

3.4.1 Reviving mon servers with the `R` command

Supermon does not automatically try to reconnect to mon servers that have died, due to its passive nature and to prevent unnecessary delay incurred by network traffic and timeout periods in reconnect attempts. The `R` command provides an explicit mechanism for clients to revive connections to mon servers that have disappeared.

When a client decides to ask supermon to retry any dead clients, it sends `R` the command. The response is an s-expression containing a list of s-expressions, one for each mon server that was dead, indicating whether the server was successfully restarted (`(0x0 revived (<IP address>) (hostname <name>))`) or is still dead (`(0x0 dead (<IP address>) (hostname <name>))`).

Chapter 4

Monhole

The *monhole* allows user-level applications to insert data into the mon data stream, enabling them to take advantage of the supermon infrastructure for fast monitoring and data collection. This feature can be used to monitor the progress of an application, monitor special (unsupported) system features such as new hardware sensor chips, and even correlate application performance with system behavior. Currently, the default `mon` does not provide monhole support. To use the monhole, `mon_with_monhole` should be run instead of `mon`. For the remainder of the chapter, we will use `mon` to mean `mon_with_monhole`.

4.1 Basic monhole functions

A user-level application (client) can insert data into the mon data stream using the monhole library (`libmonhole`). There are three basic monhole functions: `monhole_open`, `monhole_write`, and `monhole_close`. `Monhole_open` opens a connection to `mon` and initializes the connection, including sending the pound command. `Monhole_write` is used to write data to mon including changes to the pound command. `Monhole_close` shuts down the connection to `mon`. Figure 4.1 describes the arguments and return values of each of the basic monhole functions.

name	*arguments*	*return value*
`int monhole_open(monhole_t *, char *)`	monhole pound	0 on success -1 otherwise
`int monhole_write(monhole_t *, char *, int)`	monhole data string length	0 on success -1 otherwise
`int monhole_close(monhole_t *)`	monhole	0 on success -1 otherwise

Figure 4.1: Basic monhole functions: `monhole_open`, `monhole_write`, and `monhole_close`.

pound	(MONHOLE_POUND ((category1 (nr num) (field1 field2 ...)) (category2 (nr num) (field1 field2 ...)) ...))
update	(MONHOLE_UPDATE ((category1 (field1 ...) (field2 ...)) (category2 (field1 ...) (field2 ...)) ...))
event	(MONHOLE_EVENT ((category1 (field1 ...) (field2 ...)) (category2 (field1 ...) (field2 ...)) ...))

Figure 4.2: Format of monhole data for pound, update and event. `Category` is the name of a category of data, `field` is the name of each field in the category, and `nr` indicates that there are `num` number of values for each field.

4.2 Monhole data format

The monhole accepts three different types of s-expressions: *pound*, *update*, or *event*. Pound is the format of the data to be sent (*i.e.*, response to the `#` command). Update is an update of the data (*i.e.*, response to the `S` command). Event is a special data field used to indicate the occurance of some event (*e.g.*, completion of a phase of the program). Events are only sent when they occur, not with every `S` command, regardless of the mask. At this time, there can be only one event entry in pound (though there may be many events within the entry), and it must be the last entry in pound.

The format of the s-expressions is similar to that of the supermon kernel module (*i.e.*, entries in `/proc/sys/supermon`). The primary difference is that each type of s-expression (pound, update, or event) is tagged with its type. The format of the s-expressions is shown in Figure 4.2.

4.3 Testing and debugging clients

The monhole library comes with two facilities for testing and debugging clients: `fakemon` and data checking functions.

`Fakemon` is a program that emulates the monhole only. `Fakemon` provides the same connection endpoint, but prints various message (configurable) to aid in debugging the client. In addition, `fakemon` is a user-level application that does not require the `supermon` kernel module and thus can be run on any Linux machine. Monhole clients can connect directly to `fakemon` or `mon` without change.

The primary difficulty in writing monhole clients is correctly formatting the s-expressions. The monhole data checking functions verify the format of the data being sent to `mon`. These functions are also used internally by `mon` itself to protect itself (and other *upstream* clients) against poorly behaving monhole clients. There are two basic check functions (eight total for the various argument

name	*arguments*	*return value*
`int monhole_check_pound(char *, int)` `int monhole_check_pound_s(sexpr_t *, int)`	# command verbose	0 on success -1 otherwise
`int monhole_check_data(monhole_t *, char *, int)` `int monhole_check_data_ms(monhole_t *, sexpr_t *, int)`	monhole data string verbose	0 on success -1 otherwise

`int monhole_check_data_p(char *, char *, int)`
`int monhole_check_data_s(sexpr_t *, char *, int)`
`int monhole_check_data_ps(char *, sexpr_t *, int)`
`int monhole_check_data_ss(sexpr_t *, sexpr_t *, int)`
	# command
	data string
	verbose
		0 on success
		-1 otherwise

Figure 4.3: Monhole check functions: `monhole_check_pound` and `monhole_check_data`.

types): `monhole_check_pound` and `monhole_check_data`. `Monhole_check_pound` checks the given pound command to ensure that it follows the correct format. `Monhole_check_data` checks the given data (update or event) to ensure both that it follows the correct format and also that it matches the given pound command. The check functions take as arguments pointers to either `monhole_t` (for pound only), `sexpr_t`, or `char` (C "strings").

Figure 4.3 describes the arguments and return values of each of the monhole check functions. Currently, these functions are fairly simple, and do not check for duplicate categories or fields.

Chapter 5

Performance

Benchmarking supermon on specific clusters is a very important step in determining the appropriate sampling rates for clients. Perturbation in terms of memory, network, and CPU usage due to monitoring is unique to each cluster and application workload. Supermon includes a few basic programs for benchmarking both `mon`, `supermon`, and the Linux kernel module.

5.1 Benchmarking supermon

As suspected, the difference in performance for reading a fixed length data set from `/proc` versus over the network is so large that though a `/proc` benchmark is interesting, it is not very useful in practice. The sampling rates over a network are so much lower than those possible from `/proc` that a network benchmark is preferable over the single node kernel and `/proc` filesystem measurement. Additionally, network load from monitoring will drastically impact message passing applications sharing the same network resources, while moderate sampling rates of `/proc` will have significantly less effect on applications. Understanding the different factors in monitoring is important to gather meaningful information.

The network benchmark found in the `bench` directory, called `bench3` uses a simple algorithm to rapidly determine the peak sampling rate between the computer it is run on and the server it is sampling from. Supermon users should test the peak sampling rates that they can achieve between a client and `supermon`, and a client to a single `mon`. Using the `bench3` application is simple. The only required argument is the host name to connect to and test. Optional port and filter arguments are available. For example, to benchmark a node called 'n4' with a non-standard port (`2345`) and a filter, we would use the following command:

```
% bench3 n4 2345 0x3
```

The output as the program runs will show the number of samples it is testing and how long they took to complete. When it converges to a peak sampling rate, or detects two rates that it oscillates between, the minimum peak rate is returned.

5.2 Determining a good sampling rate

Depending on the data being monitored, different sampling rates are appropriate. For example, on certain mainboards monitoring hardware sensor information (such as fan speeds and motherboard temperature sensors) is not valuable at any sampling rate above 0.5 Hz. This is due to the update rate of the sensor data from the hardware. On the other hand, observing the behaviour of fast processes such as MPI job startup requires sampling rates of approximately 10-20Hz. In both cases, the rate must be chosen to capture the desired data while avoiding gathering redundant data and wasting valuable system resources. As pointed out in the Supermon paper presented at Cluster 2002, the peak sampling rate is not useful as a practical rate, but allows one to get an idea of the perturbation introduced into the system by a single sample. As the peak rate gets larger, the amount of resources required for a single sample gets smaller, thus leaving more for applications to use.

In most cases, the bottleneck that determines the peak sampling rate will be the network. This is expected due to the drastic difference between the throughput of a single computer versus a network line for a sample worth of data. By avoiding using disk for data storage (as some monitoring systems do), supermon keeps as much as possible in memory and extracts data from the kernel directly under Linux. As a result of the network-limited performance of supermon, users should be careful when determining the system configuration (*i.e.*, the structure of the tree of supermon and mon servers gathering the data to the client or front end node). The depth of the tree, defined as the maximum number of socket connections traversed from the client to a mon, is very important to consider. Since no compression or data reduction currently occurs on the path between the data source and consumer[1], each level in the hierarchy of servers will demand exactly one sample worth of resources. Therefore a single sample in a hierarchy of depth three is equivalent in terms of system resource consumption to three samples executed back to back over a single socket connection.

No standard or perfect method exists for determining a "good" sampling rate. Trial and error with educated guesses is the best one can do. Knowledge of the cluster and the applications that will be using it is necessary for determining what is best for each situation. The best suggestions we have are restated below:

- Don't sample faster than the data is updated.
- For a given metric, estimate the amount it changes naturally without monitoring. Choose a sampling rate that will minimize the amount that monitoring contributes to this value.

[1] Filtering as data is gathered to the root or client node is a planned feature in upcoming releases of Supermon.